An Index Based K-Partitions Multiple Pattern Matching Algorithm

Raju Bhukya [1], DVLN Somayajulu [2]
[1] Dept of CSE, National Institute of Technology,
Warangal, A.P, India. 506004.
Email: raju@nitw.ac.in, rajubhukya@gmail.com
[2] Dept of CSE, National Institute of Technology,
Warangal, A.P, India. 506004.
Email: soma@nitw.ac.in, somadvlns@gmail.com

***Abstract*—The study of pattern matching is one of the fundamental applications and emerging area in computational biology. Searching DNA related data is a common activity for molecular biologists. In this paper we explore the applicability of a new pattern matching technique called Index based K-partition Multiple Pattern Matching algorithm (IKPMPM), for DNA sequences. Current approach avoids unnecessary comparisons in the DNA sequence. Due to this, the number of comparisons gradually decreases and comparison per character ratio of the proposed algorithm reduces accordingly when compared to other existing popular methods. The experimental results show that there is considerable amount of performance improvement.**

***Index Terms*— Sequence, Index, Partition**

I. Introduction

Pattern matching is an important and active area of research with its varied large number of applications. DNA is the basic blue print of life and it can be viewed as a long sequence over the four alphabets *A*, *C*, *G* and *T*. The field of bioinformatics has many applications in the modern world which includes text editors, search engines, molecular medicines, industry, agriculture and Comparative biology. As the size of the data grows it becomes more difficult for users to retrieve necessary information from the sequences. Hence more efficient and robust methods are needed for fast pattern matching techniques.

Let $P = \{p_1, p_2, p_3, \ldots, p_m\}$ be a set of patterns which are strings of nucleotide sequence characters from a fixed alphabet set called $\sum = \{A, C, G, T\}$. Let *T* be a large text consisting of characters in $\sum$. In other words T is an element of $\sum^*$. The problem is to find all the occurrences of pattern *P* in text *T*. It is an important application widely used in data filtering to find selected patterns, in security applications, and is also used for DNA searching. Many existing pattern matching algorithms are reviewed and classified in two categories.

- Exact string matching algorithms
- Approximate string matching algorithms.

Exact string matching algorithm is for finding one or all exact occurrences of a string in a sequence. The problem can be stated as: Given a pattern *p* of length *m* and a string / Text *T* of length *n* (*m d” n*). Find all the occurrences of *p* in *T*. The matching needs to be exact, which means that the exact word or pattern is found. Some exact string matching algorithms are Naïve Brute force algorithm, Boyer-Moore algorithm [1], Knuth-Morris-Pratt Algorithm [2].

Inexact /Approximate pattern matching is sometimes referred as approximate pattern matching or matches with *k* mismatches/ differences. This problem in general can be stated as: Given a pattern *P* of length *m* and string/text *T* of length *n*. $(m \leq n)$. Find all the occurrences of sub string *X* in *T* that are similar to *P*, allowing a limited number, say *k* different characters in similar matches. The Edit/ transformation operations are insertion, deletion and substitution. Inexact/Approximate string matching algorithms are classified into: Dynamic programming approach, Automata approach, Bit-parallelism approach, Filtering and Automation Algorithms. Inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing.

The rest of the paper is organized as follows. We briefly present the background and related work in section II. Section III deals with proposed model *i.e.*, IKPMPM algorithm for DNA sequence. Results and discussion are presented in Section IV and followed by conclusion.

II. Background and related work

This section reviews some work related to DNA sequences. An alphabet set $\sum = \{A, C, G, T\}$ is the set of characters for DNA sequence which are used in this algorithm.

The following notations are used in this paper:
DNA sequence characters “= *{A, C, G, T}*.
ϕ denotes the empty string.
%P% denotes the length of the string *P*.
S[n] denotes that a text which is a string of length *n*.
P[m] denotes a pattern of length *m*.
CPC-Character per comparison ratio.
$\lceil x \rceil$ denotes the ceiling integer value of the real number *x*.(*i.e.*,if $x=I+f$ where $I \in N$, $0<f<1$ then $\lceil x \rceil = I+1$, $\lceil x \rceil = I$ when $f=0$).

String matching mainly deals with problem of finding all occurrences of a string in a given text. In most of the applications it is necessary for the user and the developer to be able to locate the occurrences of specific pattern in a sequence. Some of the exact string matching algorithms are available, such as Naïve string search, Brute-force algorithm, Bayer-Moore algorithm [1], Knuth-Morris-Pratt algorithms [2] .

In Brute-force algorithm the first character of the pattern *P* is compared with the first character of the string *T*. If it matches, then pattern *P* and string *T* are matched character by character until a mismatch is found or the end of the pattern *P* is detected. If mismatch is found, the pattern *P* is shifted one character to the right and the process continues. The complexity of this algorithm is *O(mn).*

The Bayer-Moore algorithm [1] applies larger shift-increment for each mismatch detection. The main difference in the Naïve algorithm had is the matching of pattern *P* in string *T* is done from right to left *i.e.,* after aligning *P* and string *T* the last character of *P* will matched to the first of *T* . If a mismatch is detected, say *C* in *T* is not in *P* then *P* is shifted right so that *C* is aligned with the right most occurrence of *C* in *P*. The worst case complexity of this algorithm is $O(m+n)$ and the average case complexity is $O(n/m)$.

The Knuth-Morris-Pratt algorithm[2] is based on the finite state machine automation. The pattern *P* is pre-processed to create a finite state machine *M* that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is $O(m+n)$. In approximate pattern matching method the oldest and most commonly used approach is dynamic programming.

Ukkonen [5] proposed automation method for finding approximate patterns in strings. The idea is to use a DFA for solving the inexact matching problem. However, automata approach doesn't offer time advantage over Boyer-Moore algorithm[1] for exact pattern matching. The complexity of the algorithm in worst and average case is $O(m+n)$. In this algorithm every row denotes number of errors and column represents matching a pattern prefix. Deterministic automata approach exhibits $O(n)$ worst case time complexity.

In 1996 Kurtz[3] proposed another way to reduce the space requirements of almost $O(mn)$. The idea was to build only the states and transitions which are actually reached in the processing of the text. The automaton starts at just one state and transitions are built as they are needed. The transitions those were not necessary will not be build. Wu. S. Manber and Myer. E[6] proposed the algorithm for approximate limited expression matching, and Wu. S. Manber.U proposed the algorithm for fast text searching which allows errors.

In the MSMPMA [7] technique the algorithm scans the input file to find the all occurrences of the pattern based upon the skip technique. By using this index as the starting point of matching, it compares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges from 1 to m-1).

In the Index based forward backward multiple pattern matching technique [4] the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches.

III. An Index based K-Partitions Multiple Pattern Matching Algorithm (IKPMPM)

The proposed work uses the indexes for the DNA sequence of character set $\sum$ to search a pattern in a string. Let *S* be a string of length *n* and the pattern *P* of length *m.* Let $\sum^*$ be the set of all possible strings with the alphabet set $\sum$. Then $S, P \in \sum^*$, $|S| = n$ and $|P| = m$. Generally $|P| \leq |S|$ *i.e.,* $m \leq n$.

By the proposed technique a table is to be built called index table, which is used to reduce the number of comparisons. Once the index table is created for a given string it is used for all types of input patterns. For each pattern *P* it checks whether the pattern *P* occurs in the sequence or not. Now choose the value of *k* (a fixed value), and divide both the string and pattern into number of substring of length *k*, each substring is called as a partition. If the value of *k* is 3 it is called 3-partition else if the value of *k* is 4 then it is 4-partition algorithm. After portioning it compares all the first characters of all the partitions. If all the characters match after searching, then it goes on to match the second character in each substring, and the process continues till the mismatch occurs or the total pattern is matched with the sequence. If all the characters match then the pattern occurs in the sequence and prints the starting index of the pattern or if any character mismatches then searching process stops and goes to the next index stored in the index table of the same row which corresponds to the first character of the pattern *P* and the above process continues .

A. IKPMPM Algorithm

Input:String S of n characters and a pattern P of m characters, where S, P∈∑* .
Output: The number of occurrence and the positions of P in S.

```
Algorithm:
Step 1: Integer arrays indexTab[4][n], charIndex[4]
Integer found:=1, n_occ:=0, n_cmp:=1, boxes;
Step 2:         FOR i:=0;i<n;i:=i+1
 indexTab[(S[i]-64)%5][charIndex[(S[i]-64)%5]++]:=i;
        End FOR
Step 3:FOR i:=0;i<chatIndex[(P[0]-64)%5]; i:=i+1
            found:=1;
            IF i+m-1 > n-1
                  found:=0
            End IF
            boxes:=⌈m/BOXSIZE⌉
            FOR k:=0;k<BOXSIZE;k:=k+1
Step 4:FOR j:=0;j<BOXSIZE*boxes;j:=j+BOXSIZE
                              IF k+j < m
                                n_cmp:=n_cmp+1
                              IF P[k+j]=S[i+j+k]
                              DO NOTHING
                                    End IF
                                    ELSE
                        found:=0
                                    End ELSE
                              End IF
                  End FOR
            End FOR
        Step 5:    IF found:=1
                   n_occ:=n_occ+1
PRINT "Pattern Found At Location i,Occurrence no is: n_occ"
        End IF
  End FOR
```

The index based multiple pattern matching algorithm uses a table (2D vector) called index table. The basic idea here is to store all the indexes of each character in its corresponding row in the 2D vector. A new technique called ASCII value based indexing technique, which is used to reduce the pre-processing time and number of comparisons. The subscript [*(S[i]-64)%5*] always returns a subscript value in the range 0,1,2 and 3 which is needed for subscripting 2D array of size *[4][n]*. The subscript values 0,1,2,3 represent the characters *T*, *A*, *G* and *C* respectively. So for each character in the string of the function (*(S[i]-64)%5)* directly references to its corresponding row in the *indexTable*. The vector *charIndex* stores the counter value of each occurrence of each character with reference to [*(S[i]-64)%5*]. For each occurrence of the first character of the pattern, the proposed algorithm compares all the characters one by one from left to right, if all characters match with the pattern it prints the pattern found from its starting index. If any character does not match it skips the search and continues to search from the next index.

TABLE I.
ARRAY SUBSCRIPT VALUES OF DNA CHARACTERS.

S.No	DNA	ASCII Value	ASCII Value-64	(ASCII Value-64)%5	Array Subscript
1	A	65	1	1	1
2	C	67	3	3	3
3	G	71	7	2	2
4	T	84	20	0	0

Let *S* be the string of length *n*, *P* be the pattern of length *m* and $S, P \in \sum^*$. Let *i* be the index of the first character of the pattern *P* in the string *S*. Let X_i be denoted as the character at the i^{th} location in the string *X*, *bs* be the fixed size for each partition and *nb* be the number of partitions(*i.e., nb*=$\lceil m/bs \rceil$) . Now for each value of *i*, for each *j=0,1,...,bs-1* and for each *j*, for each *k=0,1,...,nb-1* it checks whether $S_{i+j+bs*k} = P_{j+bs*k}$.If it is true then it will print the pattern found at the location *i* and continues the search for next value of *i*.

B. Trivial Cases in Comparisons

Case i: If $S = \phi$ *i.e.,* $|S| = 0$ and $P = \phi$ *i.e.,* $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
Case ii: If $S = \phi$ *i.e.* $|S| = 0$ and for any $|P| \geq 0$ then the number of occurrences of *P* in *S* is 0.
Case iii: If $S \neq \phi$ *i.e.,* $|S| \neq 0$ and for any $|P| = 0$ then the number of occurrences of *P* in *S* is 0.
Case iv: If $S \neq \phi$ *i.e.,* $|S| \neq 0$, $P \neq \phi$ *i.e.,* $|P| \neq 0$ and $|S| \leq |P|$ then the number of occurrences of *P* in *S* is 0.

C. This Section describes examples of the proposed approach (IKPMPM) for the DNA sequences.

Let *S=AACGACTTCAGGATCTCAGATAGCTTAGCT* be a string of 30 characters and *P=TCAGGATCTC* be a pattern of 10 characters. The following index table stores all the indexes of each character *A*, *C*, *G* and *T* in its corresponding row. The 0th row stores the indexes of occurrences of the character *T*, 1st row for *A*, 2nd row for *G* and 3rd row stores for *C*. It has chosen 4-partition IKPMPM algorithm, where the length of the pattern is 10 therefore, the number of partitions required are $\lceil 10/4 \rceil$ *i.e.*, 3.The first character in the pattern *P* is *T* so it starts the search for *P* from the 0th row (which stores the indexes of characters *T*). The first index stored in 0th row is 6 so start the algorithm from 6th character in the string and the searching process is shown below.

TABLE II
INDEX TABLE FOR A,C, T ,G CHARACTERS

T 0	6	7	13	15	20	24	25	29	
A 1	0	1	4	9	12	17	19	21	26
G 2	3	10	11	18	22	27			
C 3	2	5	8	14	16	23	28		

The algorithm first compares the first character in the pattern with the character of first index of the 0th row in table.

S=AACGAC|TTCA|GGAT|CTCA|GATAGCTTAGCT
P=|TCAG|GATC|TC |

The first character in the first partition of the pattern matches to the starting index 6 of the 0th row in *indexTab[4][20]*, so continue the matching process in the second partition. As there is a match with first character of second partition and the process continues.

S=AACGAC|TTCA|GGAT|CTCA|GATAGCTTAGCT
P=|TCAG|GATC|TC |

The first character in the third partition mismatches to its corresponding character in the string so skip the test from the starting index 6 and go to the next index available. The next index stored in *indexTab[4][20]* is 7, then start the search for *P* in *S* from the index 7 of *S*.

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT
P=|TCAG|GATC|TC |

The first character in the first of *S* is matched to the corresponding character in *P*, so continue the match for first character in the second partition.

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT
P=|TCAG|GATC|TC |

It also matches then compares the first character of the third partition of string with the corresponding character of *P* .

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT
P=|TCAG|GATC|TC |

All the first characters of each partition is matched with their corresponding character of *P*, so continue the matching process for the second characters of all the partitions.

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT
P=|TCAG|GATC|TC |

All the second characters of three partitions in *S* matches to their corresponding characters in *P*, so continue the match for all the third characters of all the three partitions.

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT

P=|TCAG|GATC|TC |

Here in the third partition there are only two characters which are present in the pattern *P*, the remaining two characters are null, so don’t need to consider them, and already compared the available two characters in the third partition. Then compare the third characters of each partition to its corresponding characters in *P*, and continue the match up to the last characters.

S=AACGACT|TCAG|GATC|TCAG|ATAGCTTAGCT
P=|TCAG|GATC|TC |

All the characters are matched from the starting index of 7. So the pattern *P* is found at the location 7 in the string *S*. The next index stored in the 0th row of *indexTab[4][20]* is 13, next search of *P* in *S* will again start from index 13 of *S*.

S=AACGACTTCAGGA|TCTC|AGAT|AGCT|TAGCT
P=|TCAG|GATC|TC |

The first character in the first partition of *S* is matched to the first character in *P*, so continue the search for the match of first character of second partition to its corresponding character in *P*.

S=AACGACTTCAGGA|TCTC|AGAT|AGCT|TAGCT
P= |TCAG|GATC|TC |

The first character in the second partition of *S* does not match with its corresponding character in *P*, so it stops the search for *P* in *S* starting from the index 13. The next location stored in the 0th row of *indexTab[4][20]* is 15, so start the search of *P* in *S* from the 15th location of *S*.

S=AACGACTTCAGGATC|TCAG|ATAG|CTTA|GCT
P=|TCAG|GATC|TC |

Clearly the first character in the first partition of *S* from the location 15 is matched with the character of *P*, so continue the matching for the first character of the second partition with the corresponding character of *P*.

S=AACGACTTCAGGATC|TCAG|ATAG|CTTA|GCT
P=|TCAG|GATC|TC |

It is mismatched, so skip the search of *P* from the index starting at 15 of *S*. The next index stored in the 0th row of *indexTab[4][20]* is 20, next continue the search from the starting index 20 of *S*.

S=AACGACTTCAGGATCTCAGA|TAGC|TTAG|CT |
P=|TCAG|GATC|TC |

Here the first character in the first partition of *S* is matched with the corresponding character of *P*, so continue the process for the second partition starting from the location 20.

S=AACGACTTCAGGATCTCAGA|TAGC|TTAG|CT |
P=|TCAG|GATC|TC |

It is mismatched, so skip the search of *P* from starting at the index 20 of *S*. The remaining indexes stored in the 0th row of *indexTab[4][20]* are 24,25 and 29. It doesn’t needs to search for *P* from these starting indexes because the length of the pattern *P* is 10, as there are no sufficient characters available to compare *P* with *S* from these indexes. It is not possible to occur the pattern from these indexes. It means the pattern *P* won’t occur form these starting indexes. Thus *P* occurred only one time in the string *S*.

D. The DNA sequence data has been taken from the Multiple Skip Multiple Pattern Matching algorithm MSMPMA [7] for testing the IKPMPM algorithm. It explains large sequence data by taking a DNA biological sequence $S \in \Sigma^*$ of size n=1024 and pattern $P \in \Sigma^*$. Let S be the following DNA sequence.

*AGAACGCAGAGACAAGGTTCTCATTGTGTCTCGCAATAGTG
TTACCAACTCGGGTGCCTATTGGCCTCCAAAAAAGGCTGTT
CAACGCTCCAAGCTCGTGACCTCGTCACTACGACGGCGAGT
AAGAACGCCGAGAAGGTAAGGGAACTAATGACGCGTGGTG
AATCCTATGGGTTAGGATCGTGTCTACCCCAAATTCTTAATA
AAAAACCTAGGACCCCCTTCGACCTAGACTATCGTATTATGG
ACAAGCTTTAACTGTCGTACTGTGGAGGCTTCAAAACGGAG
GGACCAAAAAATTTGCTTCTAGCGTCAATGAAAAGAAGTCG
GGTGTATGCCCCAATTCCTTGCTGCCCGGACGGCCAGGCTT
ATGTACAATCCACGCGGTACTACATCTTGTCTCTTATGTAGG
GTTCAGTTCTTCGCGCAATCATAGCGGTACTTCATAATGGGA
CACAACGAATCGCGGCCGGATATCACATCTGCTCCTGTGAT
GGAATTGCTGAATGCGCAGGTGTGAATACTGCGGCTCCATT
CGTTTTGCCGTGTTGATCGGGAATGCACCTCGGGGACTGTT
CGATACGACCTGGGATTTGGCTATACTCCATTCCTCGCGAG
TTTTCGATTGCTCATTAGGCTTTGCGGTAAGTAAGTTCTGGC
CACCCACTTCGAGAAGTGAATGGCTGGCTCCTGAGCGCGT
CCTCCGTACAATGAAGACCGGTCTCGCGCTAAATTTCCCCC
AGCTTGTACAATAGTCCAGTTTATTATCAAAGATGCGACAAA
TAAATTGATCAGCATAATCGAAGATTGCGGAGCATAAGTTTG
GAAAACTGGGAGGTTGCCAGAAAACTCCGCGCCTACTTTCG
TCAGGATGATTAAGAGTATCGAGGCCCCGCCGTCAATACCG
ATGTTCTTCGAGCGAATAAGTACTGCTATTTTGCAGACCCTT
TGCCAGGCCTTGTCTAAAGGTATGTTATTAATATTGACAATA
CATGCGTATGGCCTTTTCCGGTTAACTCCCTG.*

The Index Table (*indexTab[4][1024]*) for sequence *S* is very large to show here. For different patterns *P*’s the number of occurrences and the number of comparisons is shown in the Table.III for 3-partition as well as 4-partition algorithm. To check whether the given pattern presents in the sequence or not it needs an efficient algorithm with less comparison time and complexity. In general algorithms like Brute force or other conventional algorithms will take much time to do the searching process. The proposed IKPMPM technique is one simple solution which gives better performance and less complexity. This algorithm can be appreciated for decreasing the number of comparisons as well as CPC ratio .

IV. Results and Discussion

It has been observed the following in terms of relative performance of the proposed algorithm with the existing popular algorithms. From the below experimental results, improvement can be seen with the 3-partitions algorithm as well as 4-partitions. The 4-partitions algorithm gives very good performance when compared with the 3-partition technique. So it has taken different pattern sizes ranging from 1 to 16 of different DNA sequences. We have considered 4-partitions for the experimental results. As the pattern size increases the number of comparisons decreases in case of 4-partition algorithm shown in the Table III.

TABLE III
EXPERIMENTAL RESULTS OF IKPMPM ALGORITHM

S.No.	Pattern	No. of Char	No.of Occur	No.of comparison (3-patition)	No.of comparison (4-partition)
1	*A*	1	259	259	259
2	*AG*	2	53	518	518
3	*CAT*	3	11	542	542
4	*AACG*	4	5	604	614
5	*AAGAA*	5	2	635	607
6	*AAAAAAGG*	8	1	636	623
7	*TTCTTAATAAAA*	12	1	656	634
8	*GGCTGTTCAACGCTCC*	16	1	604	580

TABLE IV
COMPARISON OF DIFFERENT ALGORITHMS

Pattern	No.of occur	IKPMPM		MSMPMA		BRUTE-FORCE		TRI-MATCH		NAIVE STRING	
		No.of Com	CPC	No.of Com	CPC	No.of Com	CPC	No.of Com	CPC	No.of Com	CPC
A	259	259	0.2	1024	1.0	1024	1.0	1025	1.0	1024	1.0
AG	53	518	0.5	1230	1.2	1282	1.2	1284	1.2	1281	1.2
CAT	11	542	0.5	1298	1.2	1318	1.2	1321	1.2	1310	1.2
AACG	5	614	0.6	1359	1.3	1376	1.3	1380	1.3	1376	1.3
AAGAA	2	607	0.5	1375	1.3	1388	1.3	1393	1.3	1387	1.3
AAAAAAGG	1	623	0.6	1394	1.3	1409	1.3	1417	1.3	1407	1.3
TTCTTAATAAAA	1	634	0.6	1390	1.3	1390	1.3	1402	1.3	1399	1.3
GGCTGTTCAACGCTCC	1	580	0.5	1349	1.3	1349	1.3	1365	1.3	1349	1.3

Table. IV shows experimental results of different algorithms like MSMPMA, Brute-Force, Trie-Match, naïve-string matching with the IKPMPM algorithm. The performance of IKPMPM is observed with two parameters namely number of comparisons and comparisons per character ratio (CPC).The results show that IKPMPM provides best performance and gradually reduces over all the other methods. The number of occurrences is same for all the algorithms but the comparison and comparisons per character ratio is different for different algorithms. In IKPMPM algorithm the CPC ratio is less than 0.6 where as all the other algorithms it is greater than 1. So in proposed method the comparison per character is reduced to nearly half when compared with some of the existing algorithms.

Fig.1 shows comparisons made by different pattern matching algorithms. It is clear that the proposed (IKPMPM) algorithm outperforms when compared with all other algorithms. The current technique gives good performance in reducing the number of comparisons relative to other algorithms. The dotted line shows the IKPMPM model where as MSMPMA, Brute-Force, Trie-matching and Naïve searching is shown by solid lines.

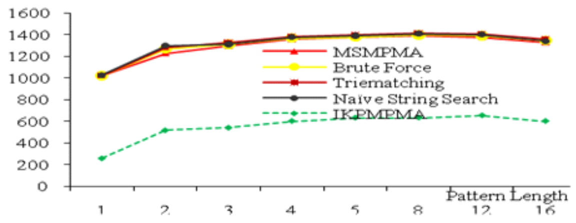

Figure 1. Comparison of different algorithms with IKPMPM.

The following are observed from the experimental results.

1.Reduction in number of comparisons.

2.The CPC ratio has reduced and is less than 1.

3.Suitable for unlimited size of the input file.

4.Once the indexes are created for sequence it doesn't need to create them again.

5.It gives good performance for DNA related sequence applications.

Conclusion

A new algorithm for pattern searching called IKPMPM is proposed. To the best of our knowledge, this paper gives the most efficient method for solving multiple pattern matching problems, till date. It is a straight approach for finding multiple occurrences of patterns from a given file. It gives very good performance when compared with other algorithms. Based on the experimental work this approach provides best performance related to the DNA sequence dataset.

References

[1]. Boyer R. S., and J. S. Moore, ''A fast string searching algorithm, '*Communications of the ACM* 20 (October 1977), pp. 762 772.

[2]. Knuth D., Morris.J Pratt.V Fast pattern matching in strings, *SIAM journal on computing*.

[3]. Kurtz. S, Approximate string searching under weighted edit distance. *In proceedings of the 3rd south American workshop on string processing. (WSP 96)*. Carleton Univ Press, 1996 156-170.

[4]. Raju Bhukya, DVLN Somayajulu,''An Index Based Forward backward Multiple Pattern Matching Algorithm, '*World Academy of Science and Technology.* (June 2010), pp. 347-355.

[5]. Ukkonen,E., Finding approximate patterns in strings J.Algor. 6, 1985, 132-137.

[6]. WU.S.,Manber U., and Myers,E .1996, A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica* 15,1,50-67, Computer Science Dept, University of Arizona,1992.

[7] Ziad A.A Alqadi, Musbah Aqel & Ibrahiem M.M.EI Emary, Multiple Skip Multiple Pattern Matching algorithms. *IAENG International Journal of Computer Science* 34:2.